

Unit 4 Data Structure

Unit 4.1 Introduction of Data Structure and application areas

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. It is a way to store and organize data so that it can be used efficiently. In other words Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. For example, Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures.

Memory Location									
200	201	202	203	204	205	206	•	•	•
U	B	F	D	A	E	C	•	•	•
0	1	2	3	4	5	6	•	•	•
Index									

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Unit 4: Data Structure

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

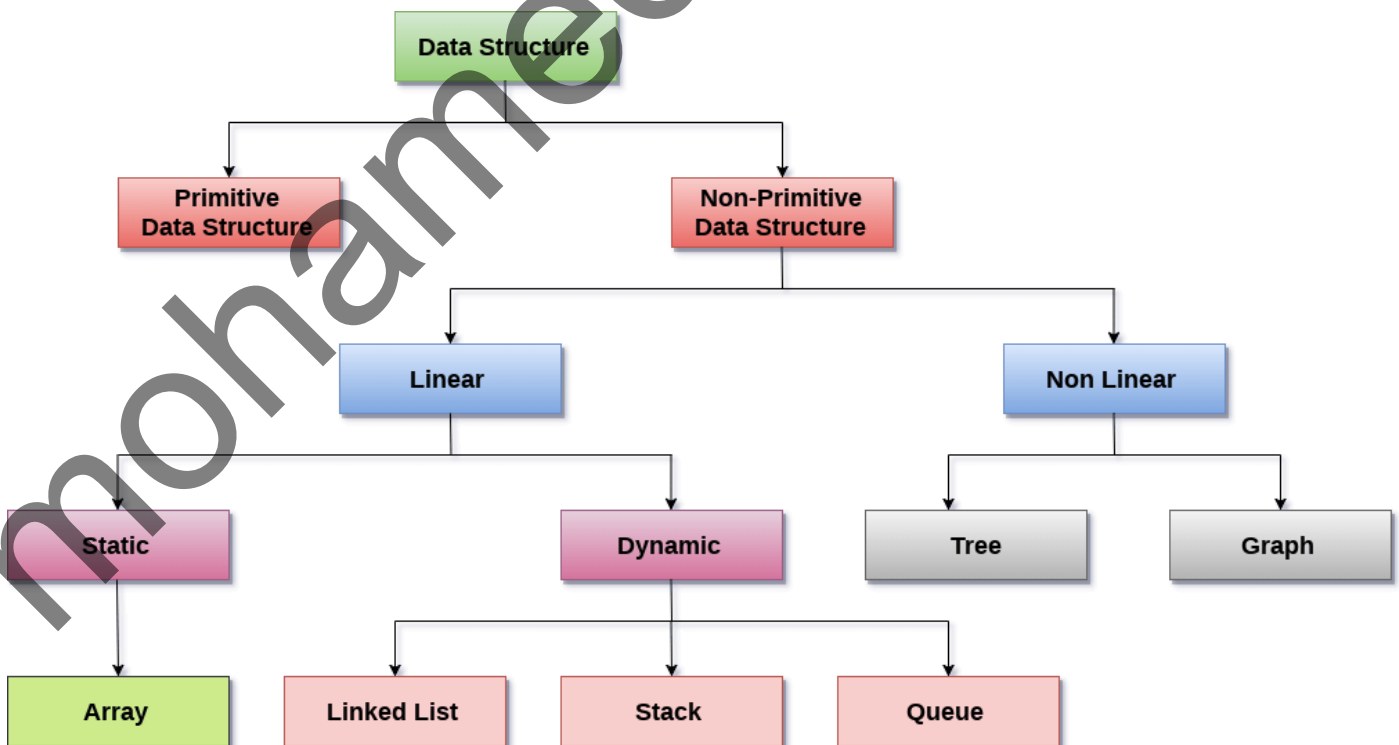
Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Areas of Application

Data structures are used in any program or software. They are used in the areas of:

- Compiler Design.
- Operating System.
- DBMS.
- Graphics.
- Simulation.
- Numerical Analysis.
- Artificial intelligence

Types of Data Structures



Unit 4: Data Structure

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array are:

age[0], age[1], age[2], age[3], ... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**. It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non - Linear Data Structure

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non - Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

Unit 4: Data Structure

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

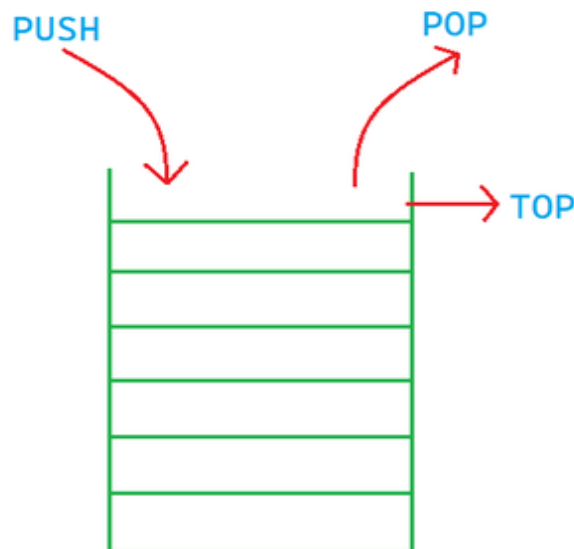
Unit 4.3 Difference among Linear and Non-Linear Data Structure

Linear Data Structure	Non - Linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure single level is involved.	In non - linear data structure multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	Its implementation is complex in comparison to inear data structure.
Data elements can be traversed in single run only.	Data elements cannot be traversed in single run.
memory is not utilized in an efficient way.	memory is utilized in an efficient way.
Examples: stack, queue, Linked list	Examples: tree, graph
Applications of linear data structures are mainly in application software development.	Applications of non - linear data structures are mainly in image processing and artificial intelligence.
Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.

Unit 4.4 Stack

Concepts of Stack(LIFO)

Stack
Insertion & Deletion
happens on the same end.



A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a **stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack**.

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Standard Stack Operations

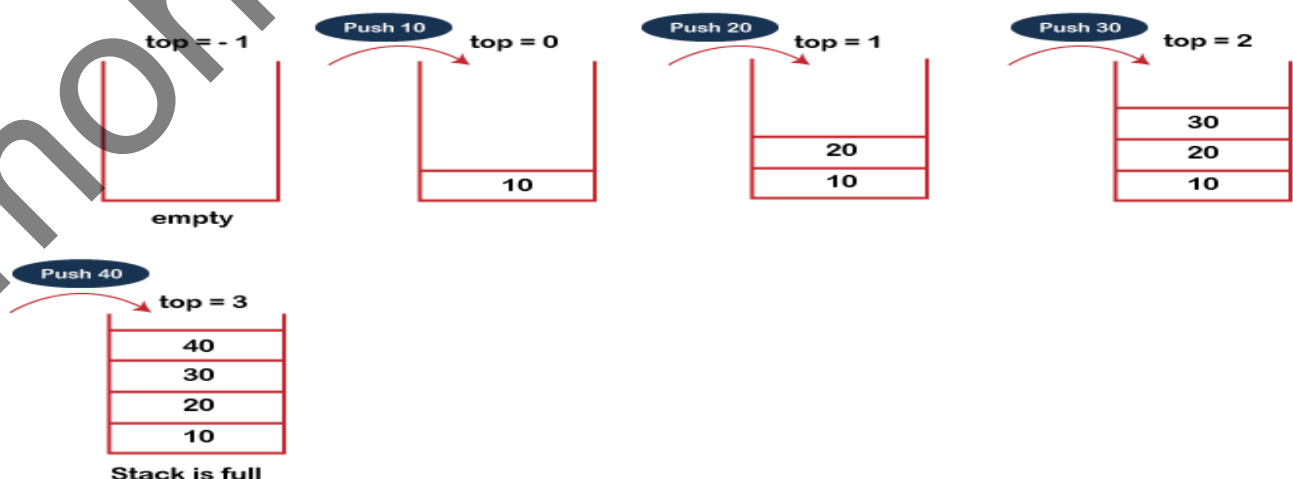
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **peek():** It returns the element at the given position.
- **display():** It prints all the elements available in the stack.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are four memory blocks in the stack; therefore, the size of the stack is 4.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 4 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 4. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 10 is entered first, so it will be removed only after the deletion of all the other elements.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.

Algorithm for push() operation

The main task of the algorithm are-

1. Check for the overflow condition.
2. Increment the **top** pointer.
3. Insert the element.

Push(value)

Here,

s = An array representing stack.

top = position of the top most element. Initially top = -1

value = data value to be inserted.

max = maximum number of elements that can be inserted in stack.

Algorithm

Step 1: [Check for the overflow condition]

If (top >= max-1) then

 Print "Overflow"

End if

return

Step 2: [Increment the top counter]

top = top + 1

Step 3: [Insert the element]

$s[\text{top}] = \text{value}$

Step 4: Exit

POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the **underflow** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the **top**
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

Algorithm for push() operation

The main task of the algorithm are-

1. Check for the underflow condition.
2. Delete the element.
3. Decrement the **top** pointer

pop()

Here,

s = An array representing stack.

top = position of the top most element. Initially top = -1

value = data value deleted.

max = maximum number of elements that can be inserted in stack.

Algorithm

Step 1: [Check for the underflow condition]

If (top == -1) then

 Print "Underflow"

End if

return

Step 2: [Delete the element]

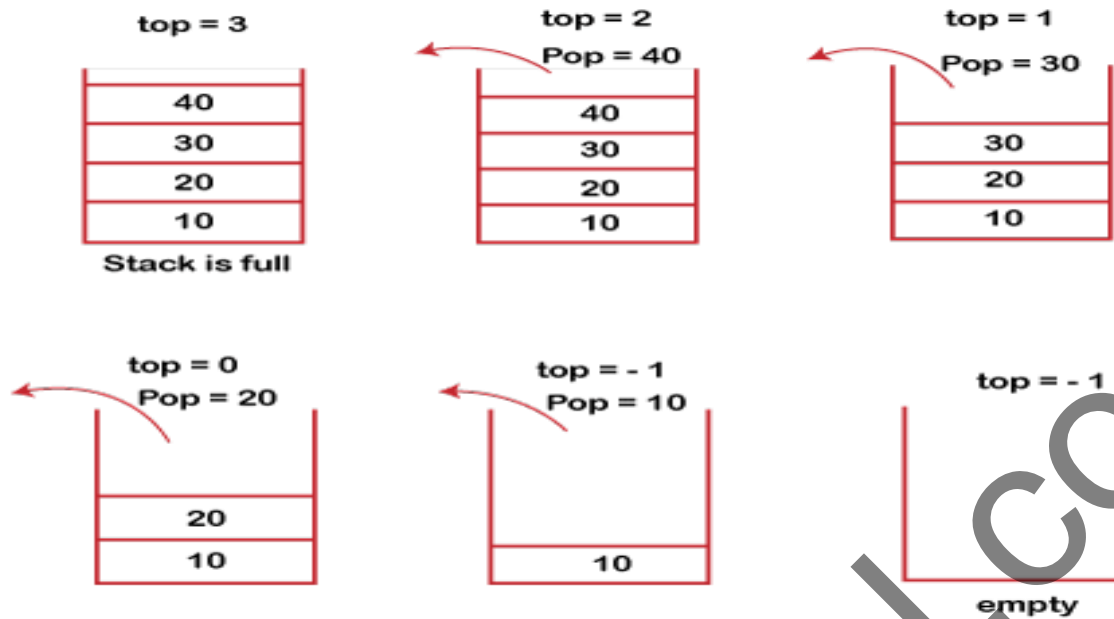
value = s[top]

Step 3: [Increment the top counter]

top = top - 1

Step 4: [Return the deleted value]

Return value



UPDATE operation

Update operation is required when value of an element **at position from the top of the stack** needs to be changed.

The steps involved in the UPDATE operation is given below:

- Before updating the element from the stack, we check whether the stack is empty.
- Before updating the element from the stack, we check whether the stack is empty at the given position.
- If the stack is not empty, we change the value of the element at **given position from the top**.

Algorithm for Update() operation

The main task of the algorithm are-

1. Check for the underflow condition.
2. Update the value of element at the given position.

Update(value)

Here,

s = An array representing stack.

top = position of the top most element. Initially top = -1

value = data value to be updated.

max = maximum number of elements that can be inserted in stack.

i = position of the element whose value needs to be updated

Algorithm

Step 1: [Check for the underflow condition]

If (top - i + 1 <= -1) then

Print "Underflow"

End if

return

Step 2: [Change the i th element from the top of the stack]

$S[\text{top} - i + 1] = \text{value}$

Step 3: Exit

PEEP operation

Peep operation is required to get a value of an element **at position from the top of the stack**.

The steps involved in the PEEP operation is given below:

- Before displaying the element from the stack, we check whether the stack is empty.
- Before displaying the element from the stack, we check whether the stack is empty at the given position.
- If the stack is not empty, we get the value of the element at **given position from the top**.

Algorithm for Peep() operation

The main task of the algorithm are-

3. Check for the underflow condition.
4. Return the value of element at the given position.

peep()

Here,

s = An array representing stack.

top = position of the top most element. Initially $\text{top} = -1$

value = data value to be returned.

max = maximum number of elements that can be inserted in stack.

i = position of the element whose value needs to be updated

Algorithm

Step 1: [Check for the underflow condition]

If $(\text{top} - i + 1 \leq -1)$ then

 Print "Underflow"

End if

return

Step 2: [Get the i th element from the top of the stack]

$\text{value} = S[\text{top} - i + 1]$

Step 3: [return the value]

Return value

Program to implement stack

```
#include<iostream.h>
#include<conio.h>
#define max 5
int s[max], top = -1;
void push()
{
    int val;
    if(top>=max-1)
    {
        cout<<"overflow";
        return();
    }
    top++;
    cout<<"enter the value to be inserted"
    cin>>val;
    s[top] = val;
    return();
}
void pop()
{
    if(top== -1)
    {
        cout<<"underflow";
        return();
    }
    cout<<"deleted value is: "<<s[top];
    top--;
    return();
}
void peep()
{
    int i;
    cout <<"enter the position"
    cin>>i;
    if(top-i+1<=-1)
```

Unit 4: Data Structure

```
{
    cout<<"underflow";
    return();
}
cout <<"the value at position " <<i<<"is: "<<s[top-i+1];
return();
}
void update()
{
    int i;
    cout <<"enter the position"
    cin>>i;
    if(top-i+1<=-1)
    {
        cout<<"underflow";
        return();
    }
    cout<<"enter the value to be updated"
    cin>>val;
    s[top-i+1] =val;
    return();
}
void display()
{
    if(top== -1)
    {
        cout<<"underflow";
        return();
    }
    For(i=top; i<=0; i--)
    {
        cout<<s[i];
    }
}
int main()
{
```

Unit 4: Data Structure

```
int ch;
do
{
    cout<<"\n1.push";
    cout<<"\n2.pop";
    cout<<"\n3.peep";
    cout<<"\n4.update";
    cout<<"\n5.display";
    cout<<"\n0.exit";
    cout<<"\n enter your choice";
    cin>>ch;
    switch(ch)
    {
        case 1: push();
                break;
        case 2: pop();
                break;
        case 3: peep();
                break;
        case 4: update();
                break;
        case 5: display();
                break;
        case 0: exit(0);
        default: cout<<"enter valid choice"
    }
}while(ch!=0);
getch();
}
```

Program to implement stack using class

```
#include<iostream.h>
#include<conio.h>
#define max 5
class stack
{
    int s[max],top;
```

public:

```
stack()
{
    top=-1;
}
void push()
{
    int val;
    if(top>=max-1)
    {
        cout<<"overflow";
        return();
    }
    top++;
    cout<<"enter the value to be inserted"
    cin>>val;
    s[top] = val;
    return();
}
void pop()
{
    if(top== -1)
    {
        cout<<"underflow";
        return();
    }
    cout<<"deleted value is: "<<s[top];
    top--;
    return();
}
void peep()
{
    int i;
    cout <<"enter the position"
    cin>>i;
    if(top-i+1<=-1)
```

Unit 4: Data Structure

```
{
    cout<<"underflow";
    return();
}
cout <<"the value at position " <<i<<"is: "<<s[top-i+1];
return();
}
void update()
{
    int i;
    cout <<"enter the position"
    cin>>i;
    if(top-i+1<=-1)
    {
        cout<<"underflow";
        return();
    }
    cout<<"enter the value to be updated"
    cin>>val;
    s[top-i+1] =val;
    return();
}
void display()
{
    if(top== -1)
    {
        cout<<"underflow";
        return();
    }
    for(i=top;i<=0;i--)
    {
        cout<<s[i];
    }
}
}
```

```
int main()
{
    int ch;
    stack s1;
    do
    {
        cout<<"\n1.push";
        cout<<"\n2.pop";
        cout<<"\n3.peep";
        cout<<"\n4.update";
        cout<<"\n5.display";
        cout<<"\n0.exit";
        cout<<"\n enter your choice";
        cin>>ch;
        switch(ch)
        {
            case 1: s1.push();
                    break;
            case 2: s1.pop();
                    break;
            case 3: s1.peep();
                    break;
            case 4: s1.update();
                    break;
            case 5: s1.display();
                    break;
            case 0: exit(0);
            default: cout<<"enter valid choice"
        }
    }while(ch!=0);
    getch();
}
```

Applications of stack

Following are the various applications of stack.

1. Recursion
2. Conversion of arithmetic expression

2.1 Infix to postfix

2.2 Infix to prefix

Unit 4.2 Recursion concepts

The function that calls itself is called **Recursion**. Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α . The function α is called recursive function.

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

- **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively. For eg., in factorial function argument value is 1 then the function does not call itself.
- **Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria. Each time the function call itself, it must be nearer to the stopping condition. For eg., in factorial, each time the function call itself, argument is decremented by one. Which causes it nearer to the stopping condition.

There are mainly two types of recursion functions.

1. Primitive recursion function: The function that directly call itself. eg Factorial.
2. Non - Primitive recursion function: The function that indirectly call itself. Eg. Ackerman's function.

Advantage:

1. It is very simple.
2. Through recursion we can solve the problem easily
3. The size of the code is reduced.

Disadvantage:

1. The system has to keep track of return addresses, parameters, and variables of each recursive call.
2. Recursive function may require large amount of memory if the depth of recursion is too large.
3. It causes system function call overhead.

Write a program to find a factorial of number using recursion.

```
#include<iostream.h>
#include<conio.h>
int fact(int);
int main()
{
    int no, ans;
    cout<<"enter the number";
    cin>>no;
    ans = fact(no);
    getch();
}
```

```

}
int fact(int n)
{
    If(n==1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}

```

Conversion of arithmetic expression

Arithmetic expression can be represented in two ways.

1. Parenthesized Expression – Infix Expression
2. Non - parenthesized Expression – prefix/postfix expression

Infix Expression – In this notation operator is placed between its two operators.

Syntax:

OP1 Operator OP2

Example

A + B, A*B

Prefix Expression – in this operator is placed before its two operands. Prefix notations are also called **Polish Notation**.

Syntax:

Operator OP1 OP2

Example

+AB, *AB

Postfix Expression – in this operator is placed after its two operands. Prefix notations are also called **Reverse Polish Notation**.

Syntax:

OP1 OP2 Operator

Example

AB+, AB*

Conversion of infix Expression to Postfix Expression

intopo (infix, postfix)

here,

infix = character array containing infix expression

postfix = character array to store resultant postfix expression

s = stack used to store operators.

Step 1: Push '(' into stack s.

Step 2: Add ')' at the end of the infix notation.

Step 3: Scan infix string from left to right and repeat step 4 to 7 un till end of string ')' is encountered.

Step 4: If operand is encountered then add it to postfix string p.

Step 5: if '(' is encountered then push it into the stack s.

Step 6: if an operator is encountered then do the following steps:

- Repeatedly pop from the stack s and add it to postfix string p, each operator from the top of the stack which has same or higher precedence then the operator.
- Push the operator into the stack s.

Step 7: If ')' is encountered perform the following steps:

- Repeatedly pop from the stack s and add it to postfix string p, each operator from the top of the stack until '(' is encountered.
- Remove '(' from the stack s. (Don't add it to postfix string p)

[End of loop at step 3]

Step 8: Exit

Examples:

Convert Expression A+B*C into postfix expression.

Step 2: A+B*C)

Infix string (character scanned)	Postfix	Stack
		(
A	A	(
+	A	(+
B	AB	(+
*	AB	(+*
C	ABC	(+*
)	ABC*+	

Ans: ABC*+

Unit 4: Data Structure

Convert Expression $A*B+C$ into postfix expression.

Step 2: $A*B+C$)

Infix string (character scanned)	Postfix	Stack
		(
A	A	(
*	AB	(*
B	AB	(*
+	AB*	(+)
C	AB*C	(+)
)	AB*C+	

Ans: $AB*C+$

Convert Expression $(A+B)*C$ into postfix expression.

Step 2: $(A+B)*C$)

Infix string (character scanned)	Postfix	Stack
		(
(		((
A	A	((
+	A	((+)
B	AB	((+)
)	AB+	(
*	AB+	(*
C	AB+C	(*
)	AB+C*	

Ans: $AB+C*$

Practice Questions:

- $A/(B-C+D)$
- $(A+B) * (C - D)$
- $(A+B) * (C - D) \wedge E * F$
- $Z + (Y * X - *W / V \wedge Y) * T * S$
- $(X * Y + Z - W) * S + (T * U / P)$
- $A / (B - C + D) * E + F \wedge G$

Write a program to convert Infix expression into postfix expression.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
char infix[50], p[50], s[50] ;
int top = -1;
void push (char ch);
char pop();
int main()
{
    int i, l, j = 0;
    cout<<"enter infix expression"
    gets(infix);
    push('(');
    l = strlen(infix);
    infix[l] = ')';
    infix[l+1] = '\0';
    for(i = 0; i <= l; i++)
    {
        switch(infix[i])
        {
            case '(':
                push('(');
                break;
            case '^':
                while (s[top]=='^')
                {
                    p[j]=pop();
                    j++;
                }
                push(infix[i]);
                break;
            Case '*':
            Case '/':
                while (s[top]=='^' || s[top]=='*' || s[top]=='/')
                {
```

Unit 4: Data Structure

```
        p[j]=pop();
        j++;
    }
    push(infix[i]);
    break;
Case '+':
Case '-':
    while (s[top]=='^' || s[top]=='*' || s[top]=='/' || s[top]=='+' || s[top]=='-')
    {
        p[j]=pop();
        j++;
    }
    push(infix[i]);
    break;
case ')':
    while(s[top]!='(')
    {
        p[j] = pop();
        j++;
    }
    top--;
    break;
default:
    p[j]=infix[i];
    j++;
    }
}
p[j] = '\0';
cout<<"postfix expression is: "<<p;
getch();
}
void push(char ch)
{
    top++;
    s[top]=ch;
}
```

```

Char pop()
{
    char c;
    c = s[top];
    top--;
    return c;
}

```

Conversion of infix Expression to Prefix Expression

intop (infix, prefix)

here,

infix = character array containing infix expression

prefix = character array to store resultant postfix expression

s = stack used to store operators.

Step 1: Push '(' into stack s.

Step 2: Add '(' at the begining of the infix notation.

Step 3: Scan infix string from right to left and repeat step 4 to 7 un till end of string ')' is encountered.

Step 4: If operand is encountered then add it to prefix string p.

Step 5: if ')' is encountered then push it into the stack s.

Step 6: if an operator is encountered then do the following steps:

- c. Repeatedly pop from the stack s and add it to prefix string p, each operator from the top of the stack which has higher precedence then the operator.
- d. Push the operator into the stack s.

Step 7: If '(' is encountered perform the following steps:

- c. Repeatedly pop from the stack s and add it to prefix string p, each operator from the top of the stack until ')' is encountered.
- d. Remove '(' from the stack s. (Don't add it to postfix string p)

[End of loop at step 3]

Step 8: Reverse the prefix expression p.

Step 9: Exit

Examples:

Convert Expression A+B*C into postfix expression.

Step 2: (A+B*C

Infix string (character scanned)	Prefix	Stack
		)
C	C	)
*	C	)*

Unit 4: Data Structure

B	CB	)*
+	CB*	)+
A	CB*A	)+
(	CB*A+	

Ans: +A*BC

Convert Expression $A*B+C$ into postfix expression.

Step 2: $(A*B+C$

Infix string (character scanned)	Prefix	Stack
		)
C	C	)
+	C	)+
B	CB	)+
*	CB	)++
A	CBA	)++
(	CBA*+	

Ans: ++ABC

Convert Expression $(A+B)*C$ into postfix expression.

Step 2: $((A+B)*C$

Infix string (character scanned)	Postfix	Stack
		)
C	C	)
*	C	)*
)	C	)*)
B	CB	)*)
+	CB	)*)+
A	CBA	)*)+
(	CBA+	)*
(	CBA*+	

Ans: *+ABC

Practice Questions:

- $A/(B-C+D)$
- $(A+B) * (C - D)$
- $(A+B) * (C - D) \wedge E * F$
- $Z + (Y * X - *W / V \wedge Y) * T * S$
- $(X * Y + Z - W) * S + (T * U / P)$

6. $A / (B - C + D) * E + F \wedge G$

Write a program to convert Infix expression into postfix expression.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
char infix[50], p[50], s[50] ;
int top = -1;
void push (char ch);
char pop();
int main()
{
    int i, l, j = 0;
    cout<<"enter infix expression"
    gets(infix);
    push('(');
    l = strlen(infix);
    for(i=l-1; i>=0;i--)
    {
        infix[i+1] = infix[i];
    }
    infix[0] = '(';
    infix[l+1] = '\0';
    for(i = 0; i <= l; i++)
    {
        switch(infix[i])
        {
            case ')':
                push('(');
                break;
            case '^':
                push(infix[i]);
                break;
            Case '*':
            Case '/':
                while (s[top]=='^')
                {
```

Unit 4: Data Structure

```
        p[j]=pop();
        j++;
    }
    push(infix[i]);
    break;
Case '+':
Case '-':
    while (s[top]=='^' || s[top]=='*' || s[top]=='/')
    {
        p[j]=pop();
        j++;
    }
    push(infix[i]);
    break;
case '(':
    while(s[top]!='')
    {
        p[j] = pop();
        j++;
    }
    top--;
    break;
default:
    p[j]=infix[i];
    j++;
    }
}
p[j] = '\0';
cout<<"prefix expression is: "<<strrev(p);
getch();
}
void push(char ch)
{
    top++;
    s[top]=ch;
}
```

```
Char pop()  
{  
    char c;  
    c = s[top];  
    top--;  
    return c;  
}
```

mohamedsohel.co.in